A Formal Model for Programming Wireless Sensor Networks

Luís Lopes[1], Francisco Martins[2], Miguel S. Silva[1], and João Barros[1]

[1]Departamento de Ciência de Computadores, FCUP, Portugal.

[2]Departamento de Informática, FCUL, Portugal.

Abstract

In this paper we present new developments in the expressiveness and in the theory of a Calculus for Sensor Networks (CSN). We combine a network layer of sensor devices with a local object model to describe sensor devices with state. The resulting calculus is quite small and yet very expressive. We also present a type system and a type invariance result for the calculus. These results provide the fundamental framework for the development of programming languages and run-time environments.

keywords: Sensor Networks, Ad-Hoc Networks, Ubiquitous Computing, Process-Calculi, Programming Languages.

1 Introduction

Developing an adequate programming model for sensor applications — involving highly dynamic networks with hundreds of power-constrained and computationally restricted nodes [3] — stands out as a formidable goal.

A well developed programming model for sensor networks, i.e. a formalism or a calculus that captures their fundamental computation and communication properties, and can serve as basis for the development of higher level programming languages, is likely to become a key enabler towards the aforementioned goal.

In the currently available implementations [4], the sensor nodes are controlled by module-based operating systems such as TinyOS [1] and are programmed using somewhat ad-hoc languages *e.g.* nesC [6] or TinyScript/-Maté [9]. Recent middleware developments such as Deluge [8] and Agilla [5] provide higher level programming abstractions on top of TinyOS, including massive code deployment where needed.

Our main contribution is a programming model with stronger formal support and analytical capabilities than the above mentioned solutions. Beyond providing a rigorous representation (or a calculus) of the sensor network at the programming level — which allows for formal verification of the correctness of programs and thorough quantification of resource usage by sensors when running programs and protocols — our model provides a global vision of a sensor network application, i.e. a specific distributed application, making it less intuitive and error prone for programmers. Moreover, we do not require the programs to be installed on each sensor individually, which would be unrealistic for large sensor networks, allowing instead for dynamic re-programming of the network.

Given the distributed and concurrent nature of sensor network operations, we build our sensor network model based on process calculi [7, 13] and also on an object calculus [2] to introduce state into the sensors. The associated theory is very rich and expressive.

Previous work on process calculi for wireless networks is scarce and does not address the peculiarities of *ad-hoc* sensor networks. In [15], Prasad established the first process calculus approach to modeling broadcast based systems. Later work by Ostrovský, Prasad, and Taha [14] established the basis for a higher-order calculus for broadcasting systems. More recently, Mezzetti and Sangiorgi [11] discuss the use of process calculi to model wireless systems. The focus of this line of work lies in the protocol layer of the networks.

In [10] we presented a preliminary version of a Calculus for Sensor Networks (CSN), devised as a two-layer calculus, offering abstractions for data acquisition, communication, and processing above the link layer of the protocol stack (i.e. without transmission errors and without packet losses). In this paper, we offer several important improvements and complementary features, most notably: (a) inclusion of state in CSN by combining it with a fundamental object calculus [2], leading to a quite simple core calculus, and; (b) a type system for the calculus and a *subject reduction* result (types are invariant under reduction).

S ::=	*Sensors*	O ::=	*Objects*
$\quad \textsf{off}$	offline	$\quad \{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$	object API
$\mid\ [\vec{P} :: O]_e^{p,r}$	sensor		
$\mid\ S \mid S$	composition	v ::=	*Values*
		$\quad b$	built-in value
P ::=	*Programs*	$\mid\ x$	variable
$\quad v$	value	$\mid\ \textsf{net}$	broadcast
$\mid\ v.l(\vec{v})$	method call	$\mid\ \textsf{loc}$	sensor object
$\mid\ v.\,\textsf{install}\ v$	API update	$\mid\ O$	object
$\mid\ \textsf{let}\ x = P\ \textsf{in}\ P$	local variable		

Figure 1: The syntax of CSN.

2 The Calculus

This section addresses the syntax and the semantics of the Calculus for Sensor Networks. For simplicity, in the remainder of the paper we refer to a sensor node or a sensor device in a network as a *sensor*. The syntax is provided by the grammar in Figure 1, and the operational semantics is given by the congruence and reduction relations depicted in Figures 2 and 3.

Syntax. Let $\vec{\alpha}$ denote a possibly empty sequence $\alpha_1 \ldots \alpha_n$ of elements of the syntactic category α. Assume a countable set of *labels*, ranged over by letter l, used to name methods within objects, and a countable set of *variables*, disjoint from the set of labels and ranged over by letter x. Variables stand for communicated values (*e.g.* basic values and objects) in a given program context.

A network is a flat, unstructured collection of sensors S combined using the parallel composition operator. The sensors are assumed to be immersed in a (scalar or vector) field and to be able to measure its intensity at their positions in space. The field describes the distribution of some physical quantity we want to model (*e.g.* temperature, pressure, humidity) in space. The position of a sensor is given in some coordinate system.

A sensor $[\vec{P} :: O]_e^{p,r}$ represents an abstraction of a physical sensing device running a sequence of instructions $\vec{P}$ and with a memory O (code plus

state). The object O is a collection of methods, the API, that the sensor makes available for internal and for external usage. Each method, $l = (\vec{x})P$, is identified by label l and defined by an abstraction $(\vec{x})P$: a program P with parameters $\vec{x}$. Method names are pairwise distinct within an object. Intuitively, the collection of methods of a sensor may be interpreted as the function calls of some tiny operating system installed in the sensor at boot time or functionalities dynamically uploaded to the sensor. The position (p) of the sensors may vary with time if they are mobile in some way. The transmission range (r) on the other hand, usually remains constant over time. In this model we abstract away from battery (e) management. In the operational semantics we simply check whether we have enough power for certain operations (*e.g.* broadcasting).

Programs are ranged over by P. A method call, $v.l(\vec{v})$, calls the method l (with arguments $\vec{v}$) in some value v. The value v may be an anonymous object, the sensor memory object, if the target is loc, or the *broadcast address*, if the target is net. In the last case, the call is broadcasted to the network neighborhood of the sensor. Installing or replacing methods in an object can be done with the construct $v.\,\mathsf{install}\ v'$, which adds the methods in v' to the object in v, eventually replacing existing implementations with new ones. In particular, this construct allows the state of objects to be modelled. The let construct allows programs to create local variables to hold intermediate values in computations. In particular, it allows the construction of arbitrarily complex data structures when combined with the appropriate methods in the sensor object.

We do not have a primitive sequential composition construct for programs. Such a construct can be easily obtained as syntactic sugar as: $\mathsf{let}\ x = P\ \mathsf{in}\ Q \equiv P;Q$ where $x \notin \mathrm{fv}(P)$. The semantics of the calculus forces the evaluation of P first and then Q exactly, since x does not occur free in Q. Thus, although we do not have a primitive sequential composition construct in the calculus, in the remainder of this paper we use this construct to impose a more imperative style of programming.

Values are the data exchanged between sensors and comprise basic values that can intuitively be seen as the primitive data types supported by the sensor's hardware, and objects that are constructed dynamically. Notice that this is not a higher-order calculus: we can only transfer the *code*, retransmit, or install objects in remote sensor.

A Simple Example. We start with a very simple ping program. We denote

$$S_1 \mid S_2 \equiv S_2 \mid S_1, \qquad S \mid \textsf{\textbf{off}} \equiv S \qquad S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3$$

(S-MONOID-SENSOR)

$$[\vec{P} :: O]_e^{p,r} \equiv [\vec{P} :: O]_e^{p,r}\{\textsf{\textbf{off}}\} \qquad \frac{e < \min(\mathsf{e}_{\mathsf{in}}, \mathsf{e}_{\mathsf{out}})}{[\vec{P} :: O]_e^{p,r} \equiv \textsf{\textbf{off}}}$$

(S-BROADCAST, S-BAT-EXHAUSTED)

Figure 2: Structural congruence for processes and sensors.

as MSensor and MSink the objects installed in any of the anonymous sensors in the network and in the sink, respectively. Each sensor has a ping method that when called broadcasts a forward call to the network with its MAC address m, and broadcasts another ping call to propagate the call in the network. The sink has a distinct implementation of this method. Any incomming call logs the MAC address given as argument. So, the overall result of the call **net**.ping() in the sink is that all reachable sensors in the network will, in principle, receive this call and will flood the network with their MAC addresses. These values eventually reach the sink and get logged.

```
MSensor(m) = { ping    = ()  net.forward(m); net.ping()
               forward = (x) net.forward(x) }
MSink      = { forward = (x) log_mac(x) }
[net.ping(), MSink] | [{}, MSensor(m1)] | ... | [{}, MSensor(mn)]
```

Semantics. The calculus has two variable bindings: the **let** construct and method definitions. The displayed occurrence of variable x is a *binding* with *scope* P both in **let** $x = P'$ **in** P and in $l = (\ldots, x, \ldots)\, P$. An occurrence of a variable is *free* if it is not in the scope of a binding. Otherwise, the occurrence of the variable is *bound.* The set of free variables of a sensor S is referred as $\mathrm{fv}(S)$.

Following Milner [12] we present the reduction relation with the help of a structural congruence relation. The structural congruence relation $\equiv$, depicted in Figure 2, allows for the manipulation of the syntactic structure of terms, making it possible for sub-terms to reduce. The relation is defined as the smallest congruence relation on sensors closed under the rules given in Figure 2.

The parallel composition of sensors is commutative and associative with **off** as the neutral element (*vide* Rule S-MONOID-SENSOR). When a sensor is

broadcasting a message it uses a conceptual *membrane* to engulf the sensors as they become engaged in communication. Rule S-BROADCAST allows for a sensor to start the broadcasting operation. An offline sensor is one with insufficient battery capacity for performing an internal or an external reduction step (*vide* Rule S-BAT-EXHAUSTED).

The reduction relation on networks, notation $S \to S'$, describes how a sensor S can evolve (reduce) to sensor S'. Reduction in a sensor occurs at the head of a sequence $\vec{P}$. In other words, in a sequence $P, \vec{P}$, the program P is running and those in $\vec{P}$ are waiting in a queue. However, we also allow for reduction within the **let** construct. In other words, the P in the example above can be of the form $\textsf{let}\ x = P'\ \textsf{in}\ P''$ and we allow the reduction *in situ* of P'. Naturally, we may have multiple levels of **let** constructs involved. For this reason we present our reduction relation using reduction contexts, or places were reduction may occur. These contexts, denoted $\mathcal{C}[\![]\!]$, are defined as follows:

$$\mathcal{C}[\![]\!] \ ::= \ [\,] \ \mid \ \textsf{let}\ x = [\,]\ \textsf{in}\ P$$

Thus, $\mathcal{C}[\![P]\!]$ denotes the process P inserted in the $[\,]$ hole of any of the above contexts.

The reduction relation is inductively defined by the rules in Figure 3. The reduction for sensors is parametric on two constants e_{in} and $\mathsf{e}_{\mathsf{out}}$ that represent the amount of energy consumed when performing internal computation steps (e_{in}) and when broadcasting messages ($\mathsf{e}_{\mathsf{out}}$).

A program P in a sensor $[\vec{P} :: O]_e^{p,r}$ may: (a) call methods in the top level object O (Rules R-METHOD-TOP and R-NO-METHOD-TOP), in anonymous objects (Rule R-METHOD), and in the network neighborhood (Rules R-BROADCAST and R-RELEASE); (b) install new methods in the top level object O (Rule R-INSTALL-TOP), and in anonymous objects (Rule R-INSTALL); (c) compute intermediate values and assign them to new variables (Rule R-LET), and (d) stop and allow another program to run (Rule R-SWITCH).

A call to a local method l with arguments $\vec{v}$ in an object O, be it the top level object or an anonymous one, such that $O(l) = (\vec{x})P$, results in the program P where the variables in $\vec{x}$ are replaced with the values $\vec{v}$. Traditionally, typed programming languages use a type system to ensure that there are no calls to undefined methods, ruling out all other programs at compile time. Our approach allows an extra degree of flexibility. When the method l is not present in object O the reduction depends on whether O is the top level object or not. In the first case we have decided to keep the

$$\frac{O(l) = (\vec{x})P \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\mathsf{loc}\,.l[\vec{v}]]\!], \vec{P} :: O]_e^{p,r} \to [\mathcal{C}[\![P[\vec{v}/\vec{x}]]\!], \vec{P} :: O]_e^{p,r}} \quad (\text{R-METHOD-TOP})$$

$$\frac{l \notin \mathrm{dom}(O)}{[\mathcal{C}[\![\mathsf{loc}\,.l[\vec{v}]]\!], \vec{P} :: O]_e^{p,r} \to [\mathcal{C}[\![\mathsf{loc}\,.l[\vec{v}]]\!], \vec{P} :: O]_e^{p,r}} \quad (\text{R-NO-METHOD-TOP})$$

$$\frac{O'(l) = (\vec{x})P \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![O'.l[\vec{v}]]\!], \vec{P} :: O]_e^{p,r} \to [\mathcal{C}[\![P[\vec{v}/\vec{x}]]\!], \vec{P} :: O]_e^{p,r}} \quad (\text{R-METHOD})$$

$$\frac{\mathrm{d}(p,p') < r \qquad e \geq \mathsf{e_{out}}}{\begin{array}{l}[\mathcal{C}[\![\mathsf{net}\,.l_i(\vec{v})]\!], \vec{P} :: O]_e^{p,r}\{S\} \mid [\vec{P'} :: O']_{e'}^{p',r'} \to \\ \qquad [\mathcal{C}[\![\mathsf{net}\,.l_i(\vec{v})]\!], \vec{P} :: O]_e^{p,r}\{S \mid [\vec{P'}, \mathsf{loc}\,.l_i(\vec{v}) :: O']_{e'}^{p',r'}\}\end{array}} \quad (\text{R-BROADCAST})$$

$$[\mathcal{C}[\![\mathsf{net}\,.l_i(\vec{v})]\!], \vec{P} :: O]_e^{p,r}\{S\} \to [\mathcal{C}[\![\{\}]\!], \vec{P} :: O]_e^{p,r} \mid S \quad (\text{R-RELEASE})$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\mathsf{loc}\,.\,\mathsf{install}\ O']\!], \vec{P} :: O]_e^{p,r} \to [\mathcal{C}[\![O + O']\!], \vec{P} :: O + O']_e^{p,r}} \quad (\text{R-INSTALL-TOP})$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![O'.\,\mathsf{install}\ O'']\!], \vec{P} :: O]_e^{p,r} \to [\mathcal{C}[\![O' + O'']\!], \vec{P} :: O]_e^{p,r}} \quad (\text{R-INSTALL})$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\mathsf{let}\ x = v\ \mathsf{in}\ P]\!], \vec{P} :: O]_e^{p,r} \to [\mathcal{C}[\![P[v/x]]\!], \vec{P} :: O]_e^{p,r}} \quad (\text{R-LET})$$

$$[P, \vec{P} :: O]_e^{p,r} \to [\vec{P}, P :: O]_e^{p,r} \qquad \frac{S \to S'}{S \mid S'' \to S' \mid S''} \quad (\text{R-SWITCH, R-NETWORK})$$

$$\frac{S_1 \equiv S_2 \qquad S_2 \to S_3 \qquad S_3 \equiv S_4}{S_1 \to S_4} \quad (\text{R-CONGR})$$

Figure 3: Reduction semantics for sensors.

call active (see Rule R-NO-METHOD-TOP). In the latter, calling an undefined method in an anonymous object causes the program to get *stuck*. We envision that if we call a method in the network after some code has been deployed (see Section 3), some sensors may receive the method call before the code is actually deployed. With the semantics we propose, the call actively waits for the code to be installed.

Sensors communicate with the network by broadcasting messages. A message consists of a remote method call on unspecified sensors in the neighborhood of the emitting sensor. In other words, the messages are not targeted to a particular sensor (there is no peer-to-peer communication). The neigh-

borhood of a sensor is defined by its communication radius, but there is no guarantee that a message broadcasted by a given sensor arrives at all surrounding sensors. Also, during a broadcast operation the message must only reach each neighborhood sensor once. Notice that we are not saying that the same message can not reach the same sensor multiple times. In fact it might, but as the result of the echoing of the message in subsequent broadcast operations. We model the broadcasting of messages in two stages. Rule R-BROADCAST calls method l in the remote sensor, provided that the distance between the emitting and the receiving sensors is less that the transmission radius ($\mathrm{d}(p, p') < r$). Each sensor that receives the call is put in the membrane associated with the emitting sensor, thus preventing multiple deliveries of the same message while broadcasting. Observe that the rule does not enforce the interaction with all sensors in the neighborhood of the emiting sensor. Rule R-RELEASE finishes the broadcast by consuming the operation ($\textsf{net}\,.l(\vec{v})$), and dissolves the membrane. A broadcast operation starts with the application of Rule S-BROADCAST, proceeds with multiple (eventually none) applications of Rule R-BROADCAST (one for each target sensor), and terminates with the application of Rule R-RELEASE.

Installing a set of methods O' in an existing object O, be it at top level or not, amounts to adding to O the methods in O' (absent in O), and to replace (in O) the methods common to both O and O'. Rigorously, the operation of installing the methods O' on top of O, denoted $O + O'$, may be defined as $O+O' = (O \setminus O') \cup O'$. The $+$ operator is reminiscent of Abadi and Cardelli's operator for updating methods in their imperative object calculus [2].

A running program P in a sequence $P, \vec{P}$, may stop its execution at any time and allow the execution of the next program in the sequence $\vec{P}$. Program P is placed at the end of the sequence, actually implementing a very simple *round-robin* interleaved execution model. This feature is essential to ensure that sensors eventually process incomming network communication. Intuitively, this rule may be seen as a very simple scheduling mechanism provided by an underlying operating system in each sensor.

3 Further Examples

In this section we present some further examples, programmed in CSN, of operations performed on sensor networks. Finally, we assume in these examples that the network layer supports *scoped flooding*. Software based *scoped*

flooding can be easily implemented with CSN although the code is a bit lengthy to present here.

Polling. In this example the sink instructs the nodes to sample the field continuously and logs the results. The sink just calls the method sample once on the network. This method propagates the call through the network and calls sample, for each sensor. The first call to sample starts by propagating the call to the network neighborhood, changes itself through an **install** call and finally calls itself to start the sampling cycle. The newly installed code of sample reads values from the field, forwards the results to the network, and calls itself again to implement a cycle.

```
MSensor   = {
  sample  = ()  net.sample();
            install {sample = ()
               let f = loc.field() in net.forward(f);loc.sample()};
            loc.sample()
  forward = (x) net.forward(x)
}
MSink     = { forward = (x) log_field(x) }
[net.sample(), MSink] | [{}, MSensor] | ... | [{}, MSensor]
```

Code Deployment. The above example assumes we have some means of deploying the code to the sensors. In this example we address this problem and show how it can be programmed in CSN. The code we wish to deploy and execute is the same as the one in the previous example. Here, we deploy the code in the network by sending an object with methods sample and forward to all the sensors. We do this by calling deploy on the network and sending the above mentioned object as a parameter. Once deployed, the code is activated with a call to sample from the sink as above.

```
MSensor   = { deploy  = (x) install x; net.deploy(x) }
MSink     = { forward = (x) log_field(x) }
[net.deploy[{
  sample  = ()  net.sample();
            install {sample = ()
              let f = loc.field() in net.forward(f); loc.sample()};
            loc.sample()
  forward = (x) net.forward(x)
 }]; net.sample(), MSink] | [{}, MSensor] | ... | [{}, MSensor]
```

$T ::=$	*Types*		
B	built-ins	$\mid$ **Net**	broadcast
$\mid \{l_i \colon \vec{T_i} \to T_i\}_{i \in I}$	objects	$\mid [l_i \colon \vec{T_i} \to T_i]_{i \in I}$	sensor object

Figure 4: The syntax of types.

4 The Type System

In this section we present a simple type system for CSN and informally discuss runtime errors.

The syntax for types is depicted in Figure 4. Types T are built from the built-in type B and the broadcast type **Net** using the constructor for object types $\{l_i \colon \vec{T_i} \to T_i\}_{i \in I}$ (and $[l_i \colon \vec{T_i} \to T_i]_{i \in I}$). Type B is the type of built-in values (*e.g.* battery, position, energy). Type **Net** is assigned to value **net** and distinguishes local method invocations from broadcast communications (via remote method invocation). A type $\{l_i \colon \vec{T_i} \to T_i\}_{i \in I}$ describes an object represented as a collection of (distinctly named) methods. Each method l_i has type $\vec{T_i} \to T_i$, where $\vec{T_i}$ is the type of the parameters of the method and T_i is its return type. For instance, type $\{\mathsf{ping} \colon \epsilon \to \{\}, \mathsf{forward} \colon B \to \{\}\}$ is the type of the object MSensor presented in Section 2. It represents an object with two methods named ping and forward. Method ping has no parameters and returns an empty object (the result from the final **net**.ping() operation). Method forward accepts a built-in value and returns $\{\}$, like ping. Type $[l_i \colon \vec{T_i} \to T_i]_{i \in I}$ denotes the type of the sensor's object. It plays an important role when typing code installation. When referring to both types of objects we use notation $< l_i \colon \vec{T_i} \to T_i >_{i \in I}$.

The type system is defined in Figures 5, 6, and 7. A typing Γ is a partial function of finite domain from variables to types. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ. When $x \notin \mathrm{dom}(\Gamma)$ we write $\Gamma, x \colon T$ for the typing Γ' such that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = T$, and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$.

Type judgements are of three forms: $\Gamma \vdash v \colon T$ means that value v has type T, under the assumptions in typing Γ; $\Gamma \vdash P \colon T$ asserts that program P has type T, under the assumptions in Γ; and $\Gamma \vdash S$ means that sensor network S is well typed, assuming the typing Γ.

The rules for typing values (Figure 5) are straightforward. Rule T-LOC

$$\Gamma \vdash b\colon B \qquad \Gamma\,, x\colon T \vdash x\colon T \qquad \frac{\forall i.\Gamma \vdash x_i\colon T_i}{\Gamma \vdash \vec{x}\colon \vec{T}} \quad \text{(T-BUILT-IN, T-VAR, T-SEQ)}$$

$$\Gamma \vdash \textsf{net} : \textsf{Net} \qquad \Gamma\,, \textsf{loc} : [l_i\colon \vec{T_i} \to T_i]_{i\in I} \vdash \textsf{loc} : [l_i\colon \vec{T_i} \to T_i]_{i\in I} \quad \text{(T-NET, T-LOC)}$$

$$\frac{\forall i \in I.\Gamma\,, \vec{x}_i\colon \vec{T_i} \vdash P_i\colon T_i}{\Gamma \vdash \{l_i = (\vec{x}_i)\, P_i\}_{i\in I}\colon \{l_i\colon \vec{T_i} \to T_i\}_{i\in I}} \quad \text{(T-OBJ)}$$

Figure 5: Typing rules for values.

$$\frac{\Gamma \vdash v\colon\ < l_i\colon \vec{T_i} \to T_i >_{i\in I} \qquad \Gamma \vdash \vec{v}\colon \vec{T_j} \qquad j \in I}{\Gamma \vdash v.l_j(\vec{v})\colon T_j} \quad \text{(T-CALL)}$$

$$\frac{\Gamma \vdash v\colon \textsf{Net} \qquad \Gamma \vdash \textsf{loc}\,.l(\vec{v})\colon \_}{\Gamma \vdash v.l(\vec{v})\colon \{\}} \quad \text{(T-BCAST)}$$

$$\frac{\Gamma \vdash v_1\colon T_1 \qquad \Gamma \vdash v_2\colon T_2 \qquad T_1 \oplus T_2 \text{ defined}}{\Gamma \vdash v_1.\,\textsf{install}\ v_2\colon T_1 \oplus T_2} \quad \text{(T-INST)}$$

$$\frac{\Gamma \vdash P_1\colon T_1 \qquad \Gamma\,, x\colon T_1 \vdash P_2\colon T_2}{\Gamma \vdash \textsf{let}\ x = P_1\ \textsf{in}\ P_2\colon T_2} \quad \text{(T-LET)}$$

Figure 6: Typing rules for programs.

assigns type $[l_i\colon \vec{T_i} \to T_i]_{i\in I}$ to the (special) variable $\textsf{loc}$. It represents the interface of all sensors and is invariant during type checking. This means that the interface for sensors is fixed by the application programmer, before type checking takes place.

As for programs, method calls are separated in local calls (rule T-CALL) and remote calls (rule T-BCAST). In a local call, method l_j must be part of the target object ($j \in I$), the type of the arguments must agree with the type of the parameters ($\vec{v}\colon \vec{T_j}$), and the type of the invocation (T_j) is the return type of the method. A remote call ($v\colon \textsf{Net}$) is type checked as a local call, apart from its return type that is always the empty object ($\{\}$), meaning that the return value of a remote call is ignored. Code installation (rule T-INST) is allowed either in anonymous objects, or in the sensor's object. The definition of operation $T_1 \oplus T_2$ is similar to that of operation $+$ for combining objects, but has only a meaning for $[\_] \oplus [\_]$, $[\_] \oplus \{\_\}$, and $\{\_\} \oplus \{\_\}$. Allowing

$$\frac{\Gamma \vdash O\colon \{l_i\colon \vec{T_i} \to T_i\}_{i\in I} \qquad \Gamma \vdash \vec{P}\colon \_ \qquad \Gamma \vdash pre\colon \vec{B} \qquad \Gamma \vdash \textsf{loc} : [l_j : \vec{T_j} \to T_j]_{j\in J} \qquad I \subseteq J}{\Gamma \vdash [\vec{P} :: O]_e^{p,r}} \quad (\text{T-SENSOR})$$

$$\Gamma \vdash \textsf{off} \qquad \frac{\Gamma \vdash [\vec{P} :: O]_e^{p,r} \qquad \Gamma \vdash S}{\Gamma \vdash [\vec{P} :: O]_e^{p,r}\{S\}} \quad (\text{T-TERM, T-BSENSOR})$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1 \mid S_2} \qquad \frac{\Gamma \vdash P\colon \_ \qquad \Gamma \vdash \vec{P}\colon \_}{\Gamma \vdash P, \vec{P}\colon \_} \quad (\text{T-PAR, T-SEQP})$$

Figure 7: Typing rules for sensor networks and for program sequences.

$\{\_\} \oplus [\_]$ may cause an anonymous object to refer to undefined methods, since it inherits the complete interface from the type of the top level object. The result of an **install** operation is the altered object.

Regarding the typing rules for sensors, we focus on rule T-SENSOR, as the remainder of the rules should be simple to follow. When typing a sensor we make sure that the methods available in the sensor's object conform with the global, network-wide, interface defined by **loc**. Notice that a sensor may offer just a subset of the interface methods ($I \subseteq J$), since some of them may not yet be available (installed) in the sensor.

The following result ensures that types are preserved during reduction.

Theorem 1 (Subject Reduction) *If* $\Gamma \vdash S$, $S \to S'$, *then* $\Gamma \vdash S'$.

The proof proceeds by induction on the derivation tree for the reduction $S \to S'$ and is a straightforward case analysis; due to space constraints we omit it, as well as the standard intermediate results required for the proof.

5 Conclusions and Future Work

Based on our formal computational model for programming sensor networks (presented in [10]), we developed a natural extension to account for sensor nodes with different states. In addition, we introduced a static type system that enables safe programming of sensor networks. It is worth pointing out that typed sensor network applications can be filtered at compile time, allowing for premature detection of certain types of programs that would

produce run-time errors. As a first step towards proving the *type safeness* of the model, we provided a *subject reduction* result.

It is our belief that these results set the basis for a range of programming language idioms for sensor networks which we aim to implement in the immediate future. Ultimately, we would like to program real-world sensor network applications using such a language and its associated run-time system.

References

[1] The TinyOS Documentation Project. Available at `http://www.tinyos.org`.

[2] M. Abadi and L. Cardelli. An Imperative Object Calculus. In *TAPSOFT '95: Theory and Practice of Software Development*, number 915 in LNCS, pages 471–485. Springer-Verlag, 1995.

[3] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[4] D. E. Culler and H. Mulder. Smart Sensors to Network the World. *Scientific American*, 2004.

[5] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662. IEEE, June 2005.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[7] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the ECOOP '91 European Conference on Object-oriented Programming*, LNCS 512, pages 133–147. Springer-Verlag, 1991.

[8] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd*

international conference on Embedded networked sensor systems, pages 81–94. ACM Press, 2004.

[9] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[10] M. S. Silva and F. Martins and L. Lopes and J. Barros. A Calculus for Sensor Networks. available from `http://arxiv.org/abs/cs.DC/0612093`, December 2006.

[11] N. Mezzetti and D. Sangiorgi. Towards a Calculus for Wireless Systems. In *Proc. MFPS '06*, volume 158 of *ENTCS*, pages 331–354. Elsevier, 2006.

[12] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[14] K. Ostrovský, K. V. S. Prasad, and W. Taha. Towards a Primitive Higher Order Calculus of Broadcasting Systems. In *PPDP'02, International Conference on Principles and Practice of Declarative Programming*, 2002.

[15] K. V. S. Prasad. A Calculus of Broadcasting Systems. In *TAPSOFT, Volume 1*, pages 338–358, 1991.